# Processing Graph Method 2.1 Semantics

by
David J. Kaplan & Richard S. Stevens

July 29, 2002

## 1 Introduction

The purpose of this document is to specify the Processing Graph Method (**PGM**). We will tell you what problem PGM is supposed to help solve. Then we will tell you what PGM is.

Specifying PGM is a complicated proposition because PGM has many related parts. We will tell you in a general way, what the parts are, how they fit together, and how certain assembled parts (*processing graphs*) of PGM operate. Then we will describe how *command program procedures* are used to help build and control *graphs*. Finally we list and describe the functionality of the *command program procedures*.

For the remainder of this document, we shall shorten the phrase “processing graph” to “graph” whenever we can do so without confusion.

## 2 What problem was PGM built to solve?

The increasing need for computing power and higher throughput has led to the development of computer architectures with multiple and sometimes disparate processors operating as a network. Different architectures have different inter-processor communication layouts. Using current methods to write application programs that run quickly on a multi-processor machine requires the programmer to be very familiar with the hardware and system architecture of the target machine. The program must be tuned to take advantage of the features of the target architecture. As a result, transporting a program from one architecture to another requires much more than recompiling the program’s source code. A program written for machine architecture **A** cannot, in all likelihood, be compiled for machine architecture **B**. And if **A** can be compiled for architecture **B**, it will not be tuned for architecture **B** and is not likely to achieve high throughput.

The situation is similar to what existed in the early days of single processor machines, which differed in their instruction sets. Initially, machines were programmed in machine language, and then in assembly language. A program written for one machine had to undergo a complete rewrite to run on another machine. High-level languages - and compilers for them - were invented to make it possible to write a program once. To run a high-level language program on a new machine, one recompiled that program using a compiler written for the new machine.

Because software could then be transported easily from one machine to another, rewrites of application programs each time a new machine appeared on the market became less necessary. Moreover high-level languages were easier to write and to read. People with less training could write and maintain programs and rely less on micro-code and assembly language programmers. The combined effect of easy portability between machines and accessibility to the general technical public not only reduced the cost of software, it also broadened the market. This resulted in the enormous growth of computer technology that has occurred over the last few decades. A similar breakthrough is needed today for multiprocessor architectures. We must make it easy to write programs and to move them from one machine to another. This breakthrough will require the development of an architecture independent language, together with tools that support the affordable development of compilers for that language - compilers that target multiprocessor architectures.

Because of the sequential syntax and semantics of high-level languages, programs written in a standard high-level language obscure concurrency. Programs written using PGM are iconic and have internal processes that have been organized in a way that display the application’s inherent concurrency. PGM programs are independent of how and at what moment data is moved among those processors. PGM also provides control mechanisms for responding to the changing situations that are part of most dynamic military and civilian situations.  Because the details of how to handle data movement devices are not in the application specification, those details must be built into the operating system.

## 3 A View of PGM

A PGM application is built of two components: *a command program* and *a graph*. Each of these components has access to a properly configured environment that includes compilers, an operating system, a set of PGM Libraries, and high-level languages such as ANSI C, C++, Java, Ada 83, or Ada 95.

### 3.1. Families

A *family* in PGM extends the notion of a finite array. Families are used extensively in PGM.

#### 3.1.1. Description of a family

A family is built on a single *type* called the *base type*. The base type elements in a family are called *leaves.* All leaves in a family have the same base type. As with a multi-dimensional array, identification of each leaf in a family requires an index for each level in the family. The number of indices needed to identify a leaf is called the *height* of the family. Thus, for example, a family with height 2 is like an array with two dimensions and needs two indices to identify each leaf.

A family with height zero is, by definition, a leaf of the specified base type and needs zero indices to identify that leaf. A family with height ***n*** > 0 consists of zero or more *children*, each *child* being a family with height ***n-1***. A family with non-zero height and no children is called *empty*. We will use the symbol, ,to represent $\boxed{\text{content = 0}}$ an empty family. The location of its instance within the family allow us to infer the height of the instance.

The number of children of a family with non-zero height is called the *content* of the family. If **X** is a family with non-zero height, then it is not required that all children of **X** have the same content. A family with height 0, by definition, has no children and exactly one leaf

Each child of a family is identified by an integer index. The smallest index is called the *lower bound*. If **LB** is the lower bound (any integer value) and **N** ≥ 0 is content, then the indices for the children are **LB, LB+1**,…, **LB+N-1**.

Let ***n*** > 1. Let **X** be a family with height **n** and one or more children that are all empty. Let **Y** be an empty family with height ***n***. We note that although both **X** and **Y** have zero leaves, they are different, because **X** is not empty and **Y** is empty

The type of a family is its height together with its base type. Thus an empty family with height two and another empty family with height three have different types, even if they have the same base type. If **T** is a given base type, we may speak of a "family of **T**" or a "**T** family" to mean "a family with base type **T**."

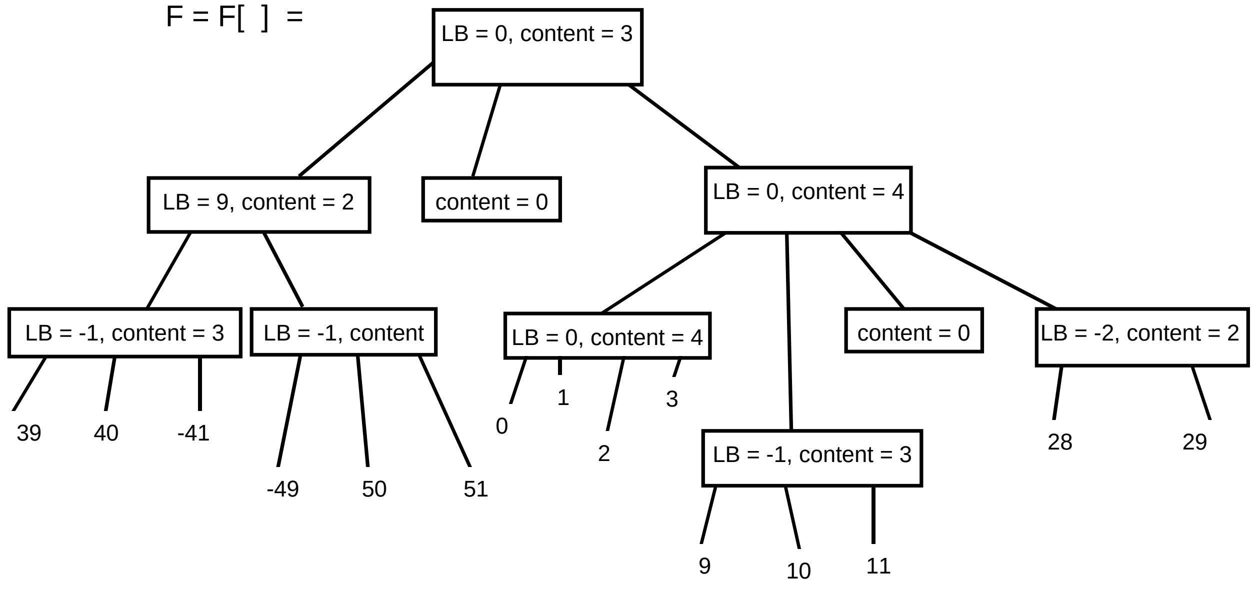
Each family within a graph may have a user-supplied name. To identify a leaf in such a family requires both the family name and the indices. We will speak of an *array of family indices* or an *array of indices*.

Note that a family is a rooted tree with some additional structure. Each branch of the tree has a specified distance from the root. All leaves have the same distance from the root that is equal to the height of the family (hence the motivation for our use of the word leaf). Each branch that has distance 1 from the root is a child of the family. These branches are identified by the values of an index ***i*** such that
**LB ≤ i** < **LB+N-1**.

### 3.1.2. An Example of a family

As we have said families are ubiquitous in PGM; and for that reason we will show here an example of a family that meets these definitions.. We choose to name the family F.

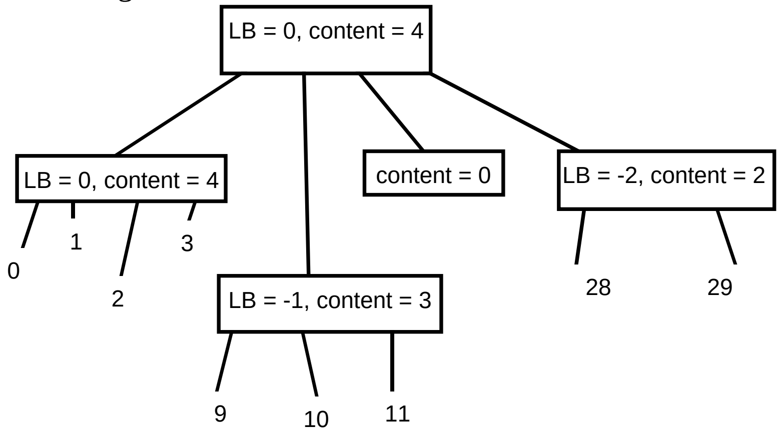
F = F[ ] =



Notice the height of F is 3.

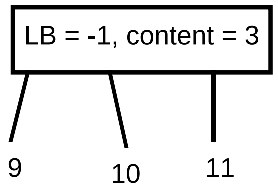
We compute F[2] obtaining:

F[2] =



Notice the height of F[2] is 2.

F[2,1] =



Notice the height of F[2, 1] is 1.

Finally compute F[2,1, 0] obtaining:

F[2,1, 0] = 10

Notice the height of F[2,1 , 0] is 0 and, in fact, 10 is a leaf.
Thus the element of F with indices [2][1][0] has value 10.

Some constructs have associated families. If that is the case, we may systematically refer to a specified one of these families as the construct's family. We will see that families are used extensively in PGM. Specifically we will find that such graph objects as transitions, queues, graph variables, graph ports, and transition ports and such data objects as queues may contain a family of tokens and the structure of a given token may contain a family.

## 3.2. Graphs and Comand Programs

A graph consists of *nodes* and *directed arcs*. All the nodes in a graph have user-supplied names. In a graph, data exists in the form of *tokens* of specified types. The tokens move, one after another, from node to node in paths whose direction is defined by directed arcs. Directed arcs have no names.

Nodes are of two *categories*. A node is categorized as a *place*, which represents data storage, or as a *transition,* which represents the transformation of data.

The places of a graph may be initialized. The words "transition" and "place" are adapted and modified from the concept of Petri Nets. As in Petri Nets, arcs are only allowed between nodes of differing category.

The topology of a graph describes which node is connected to which node. This process indirectly specifies how many nodes there are in the graph.

Two graphs are *isomorphic* if there exists a one-to-one mapping between the nodes and directed arcs of one graph to the nodes and directed arcs of the other graph that preserves the direction of the corresponding directed arcs, and category of corresponding nodes

The state of a graph is what tokens of the graph are in what places of the graph.

*Command programs* are programs written in a high-level language. Command programs use procedures called *command program procedures*, which reside in a library called the *Command Program Procedure Library*. Command program procedures manipulate graphs. Command programs use command program procedures to create and manipulate graphs. Command programs support interaction with operators to provide a set of choices for processing, thus providing an interactive interface between an operator and graphs.

## 4 Instantiation Procedures

Before we can run a graph we must create it. The principal way of creating a graph is to write a procedure that when run, will create the graph. To create a graph, the PGM user describes an iconic form of a graph by using a *graphic user interface* (**GUI**). Using click-drag-and-drop techniques, the user puts icons that represent graph constructs on a terminal screen. Each icon has an associated window that contains detailed information about the object represented by that icon. The user may edit the information in this window by completing tables and supplying collateral information. When the graph is fully specified, upon user command, the GUI editor automatically writes a procedure that will be called by a command program, which, upon execution, will create the graph. These procedures are called *instantiation procedures* (**IPs**). We say that the IP *instantiates* the graph.

Alternatively, a user may bypass the GUI editor, and directly write instantiation procedures.

The command program calls an IP to construct a graph called the *main graph.* An IP may be written to call another IP, creating a graph to be included in the graph under construction. Such a graph is called an *included graph.* The main graph and any included graph may have any number of included graphs. Henceforth in this section we refer to the main graph, together with all its included graphs as the graph.

When an IP completes its execution, all transitions of the graph are blocked from starting execution. The detailed structure of the graph will be determined by the values of parameters called *graph instantiation parameters* (**GIPs**) that are inputs to an IP at the time the IP is called.

Because the topology of the graph depends on the values of GIPs, graphs instantiated by different calls to the same IP may not be isomorphic. For example, a GIP may be used to specify the number of nodes in a family of nodes

The graph, after being created, may be transformed to improve its performance before it is run. The nodes of the graph may be automatically distributed among the processors of a multiprocessor architecture in a manner to effectively balance the throughput and latency of the application.

After this improving transformation, an unblocking procedure will unblock the transitions of the modified graph.

IPs use command program procedures called *graph construction procedures* (**GCPs**) that have the capabilities of constructing and connecting the nodes that constitute a graph. The command program constructs the graph by calling its IP. After the graph is constructed we may start the graph. Later we may *suspend* the execution of the graph and either resume execution of the graph or delete the graph.

## 5 Anatomy of the graph

In this section we use the word graph to mean a main graph or an included graph

### 5.1. Graphs and graph parts

As we have said, a graph consists of nodes and directed arcs. All the nodes in a graph have user-supplied names. In a graph, data exists in the form of tokens of specified type. The tokens should be thought of as moving in order from node to node in paths whose direction is defined by directed arcs. Directed arcs have no names.

A token is a family. The type of a token is its height together with its base type. Its family tree and the values of its respective leaves determine the value of a token. The base type of a token may be a standard variable type in the high-order language, like integer or float, or it may be a user-defined type.

The tokens enter and leave nodes by means of the node's *ports*. A *port's mode* is the type of the entering or leaving tokens. The mode of a port is unchanging. All the ports in a *port family* have the same mode. A port family has a name that is derived from the name and structure of the node that owns the port family. To connect a port to another port, the connection is indicated with a directed arc pointing from one port to the other port.

In the GUI, one node icon may be used to represent a *node family*. One *included graph icon* may be used to represent a family of included graphs. One directed arc may be used to represent all the connections between two port families. Two ports connected by a directed arc have the same mode.

The following text is predicated on **Figure 1**. Let **P1** be a port of node **N1** and **P2** be a port of node **N2**. Let a directed arc point from **P1** to **P2**; then **P1** is called an *output port* (of **N1**) and **P2** is called an *input port* (of **N2**). We say that **P1** and **P2** are *linked*. We say that **N1** is an *input node* of **N2** and that **N2** is an *output node* of **N1**.

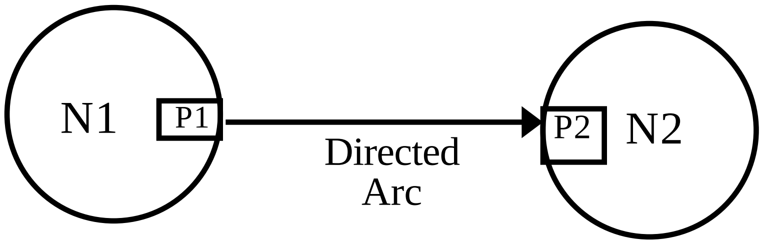


Figure 1

If **N1** is a place and **N2** is a transition, then **N1** provides stored data to **N2** so that **N2** may process it. If **N1** is a transition and **N2** is a place, then **N1** provides processed data to **N2** so that **N2** may store it.

Each place has a specified mode, which determines the type of all the tokens stored in the place. Each place has one data input port family and one data output port family; the mode of these ports is the same as the mode of the place.

If **N1** is a place and **P1** is a data output port of **N1**, then **N1** is said to be a data input place of **N2**. If **N2** is a place and **P2** is a data input port of **N2**, then **N2** is said to be a data output place of **N1**.

Some of the unconnected input and output ports of nodes within a graph may be called graph input ports or graph output ports.

An input port of a graph is identified with an input port of a node in that graph. Similarly, an output port of a graph is identified with an output port of a node in that graph.

Each node is contained in a node family. Each port of a node is contained in a port family of the node. Each node family and each port family has a family name.

## 5.2. Place

Each place has an associated family whose children are tokens. Each place may be initialized with zero or more tokens. The *content* of a place is the content of its associated family of tokens. Each place also has an associated family of *input data ports* whose family name is **INPUT** and an associated family of *output data ports* whose family name is **OUTPUT.** .Still referring to **Figure 1**, suppose that the *node families* containing **N1** and **N2** and the port families containing **P1** and **P2** all have height zero (i.e., they are single nodes with single ports). Then if **N1** is a place whose name is "**name1**" then **P1's** name is "**name1.OUTPUT**". If **N2** is a place whose name is "**name2**", then **P2's** name is "**name2.INPUT**". If the family containing a node or port has height > 0, then the full name of the node or port contains the indices needed to identify the node or port. The syntax to express the indexing is implementation specific.

No more than one directed arc may connect a given place port to transition ports. No more than one input port of a place and no more than one output port of a place may be connected to ports of the same transition.

The kinds of place are *queue* and *graph variable*.

Each place has an integer attribute called *capacity*. No transition's execution may result in any of the transition's data *output place's content* exceeding the data *output place's capacity*. Depending on the kind of place, the capacity may be changed by the system. The capacity of a place may not be changed directly by the values of data tokens in the graph or by any direct action of the command program.

The words *produce* and *consume*, for the purposes of PGM, are technical words.
*Produce* refers to the storing of new tokens in a place.
*Consume* refers to the possible removal of tokens from a place.
The meanings of produce and consume are modified by the kind of place in which the tokens might be stored or removed.

### 5.2.1. Queue
Each *queue* is a place that during graph execution
- sends out tokens from its data output ports in a first-out manner;
- takes in tokens through its data input ports in a first-in manner; and
- stores tokens in the queue's associated family.

The effect of consuming **N** tokens from a queue is to remove **N** tokens from the queue. Unless otherwise stated, the first **N** tokens are removed.

If the input port family of a queue has more than one port, then tokens are stored in the queue in the order that they are produced to the queue. This order depends in part on the order in which the input transitions execute.

If the output port family of a queue has more than one port and an output transition executes, then the token(s) read and consumed from the queue are determined by the state of the queue when the transition executes and the number of tokens being read or consumed. No other output transition may begin execution until the currently execution is complete.

### 5.2.2. Graph variable
Each *graph variable* is a place that stores exactly one token. The content of a graph variable is one. The capacity of a graph variable is one.

A graph variable's associated token family has one initial token. During *graph execution*, each token produced to a graph variable's associated family through one of its data input ports replaces the token currently stored in the graph variable.

During execution of the graph, a graph variable will behave as if it has unlimited content and unlimited capacity. The result of reading **N** tokens from a graph variable is **N** tokens, all with the same value as the currently stored token. The result of producing **N** tokens to a graph variable is the storage of the last token, replacing the previously stored token. The effect of *consuming* **N** tokens from a graph variable is nil.

If the family of input ports of a graph variable has more than one *input port*, and two *input transitions* execute, then both tokens are produced to the graph variable in an order that may depend on the order in which the two input transitions execute. Each new token replaces the previously stored token. Any *output transition* of the graph variable will, while executing, read the currently stored token.

## 5.3. Transition

PGM supports user created transitions. PGM supplies the capabilities that allow the user to build very complicated transitions. A user may build transitions that are beyond our ability to automatically analyze and improve. We will describe PGM's full capability and then we will describe a constrained capability that we believe will allow adequate transitions to be built but will limit transition complexity sufficiently to allow automatic analysis and improvement.

First we will provide a bit more general information about transitions.

If a transition has an output port that is not connected to an input port of a place, then that output port does not block execution of the transition. Any tokens produced at an unconnected transition output port are discarded.

Unless otherwise stated, if a transition has an input port that is not connected to an output port of a place, then that input port behaves as if it were connected to the output port of an *empty queue*.

A graph is defined to be *quiescent* if and only if for each of its transitions, at least one of that transition's input places has insufficient tokens, or at least one of the transition's output places will not accept tokens.

### 5.3.1. The unconstrained transition

Now let us look at a transition's undisciplined capability.

During the process of a transition executing, the transition reads tokens from a computed choice of the transition's data input places. The transition calculates again the transition's data input places and continues the process of reading tokens, etc. After an unpredictable sequence of operations, based on the tokens read and the transition's makeup, the transition's execution will be complete. (If we are unlucky, the sequence may never terminate.)

If the sequence does terminate, then a computed number of computed tokens is attempted to be placed on to the transition's data output places within the constraint that no transition's execution may result in any of the transitions data output places' content exceeding the data output place's capacity. Otherwise the transition decides what to do.

This decision may be to terminate execution having neither produced nor consumed any tokens.
If the capacities are met, then some computed numbers of tokens are consumed from the transition's data input places.

By what means does the transition make the calculations we just described? The answer is the *transition statement*.

### 5.3.2. Transition statement

Each transition has an associated underlying statement called a transition statement, which specifies what the transition does during transition execution.

A transition statement is a control structure in the supporting language. Multiple statements, connected by the appropriate delimiters, will constitute *a* transition statement.

Such statements as:
- assignment statements,
- procedure calls,
- if-then-else statements,
- case statements,
- while statements,
- repeat statements,
- for statements, and
- do statements,

are likely to be available in the supporting language.

It is more apropos to say what is not allowed in a transition statement. These forbidden actions are those that create programs, procedures, functions, classes, types, and constructors. It is acceptable to call a procedure or to create a variable of an already constructed class or type in a transition statement. None of the constraints mentioned here applies to a procedure called within a transition statement. Any procedure that is called within a transition statement is called a *primitive*.

Let **T** denote a transition.

All variables that are read or written by executing **T** are either:
related to place ports which are connected by directed arcs to ports of **T**
or
local to **T's** and only **T's** transition statement.

All variables that are read by executing **T** may not be written except by executing **T**.

From one execution to the next, all variables that are referred to within a transition statement and any primitives that are called within the transition statement shall reinitialize their state.

### 5.3.3. The constrained transitions
We divide transitions into two groups, *ordinary transitions* and *special transitions.*

#### 5.3.3.1. The ordinary transitions
**Brief Description**: The transition executes when for each data input place there is an available token which is read. The transition statement computes using the read tokens, and for each data output place a token is produced, after which a token is consumed from each data input place.

**Detailed Description**:
An ordinary transition may execute if for each place **P** with a data output port that is connected to an input port of the transition, **P** is not empty.

For each place **P** with a data input port that is connected to an output port of the transition, the transition's execution would not result in **P** exceeding its capacity.

An ordinary transition is executed as follows:
- For each input port **P** of the transition, a token is read from the place whose output port is connected to **P**.
- If the place is a queue, the first token on the queue is read;
- The transition statement, including any primitives, executes;
- A token is produced to each place connected to an output port of the transition;

- For each place with a data output port that is connected to an input port of the transition, a token is consumed.

Let **T** be an ordinary transition, and let **P** be an input port family of **T**. It is required that all ports in the family **P** have the same mode, i.e., all tokens read via ports of **P** must have the same height and base type. For **T** to be ready to execute, it is required that every port of **P** be connected to a place containing at least one token.

Let **H0** be the height of the port family **P**. Let **H1** be the height of the common mode of **P**. When **T** executes, all of the tokens read via the ports of **P** are assembled into a single token whose height is **H0 + H1**, using the family tree of the port family **P**. The transition statement operates on the resulting assembled token.

Let **T** be an ordinary transition, and let **P** be an output port family of **T**. It is required that all ports in the family **P** have the same mode, i.e., all tokens produced via ports of **P** must have the same height and base type. For **T** to be ready to execute, it is required that every port of **P** be connected to a place that can accept at least one token.

Let **H0** be the height of the port family **P**. Let **H1** be the height of the common mode of **P**. When **T** executes, the transition statement must produce a single token **K** for the port family **P**. The height of this token must be **H0 + H1**, and the top **H0** levels of its family tree must be compatible with the family tree of the port family **P**. The token **K** is then disassembled to produce a token with height **H1** via each respective output port of **P**. This disassembly is the reverse process described above for an input port family.

#### 5.3.3.2. Special transitions of the transitions data input places

The *special transitions* are *pack* and *unpack*. Pack takes in some specified number of tokens, which may differ from execution to execution, and packs them into a single token. Unpack takes in a single token and unpacks it into a stream of tokens.

#### 5.3.3.2.1. Pack

**Brief description:** Each pack has an input place from which a specified number of tokens is read. Pack has an output place via which a single token is produced. The tokens read are determined by the values of tokens read from four input ports called READ, READOFFSET, CONSUME, and CONSUMEOFFSET. The single token produced is a family consisting of the tokens read.

**Detailed description:**
**Input Ports*:***
- an input port named INPUT of any mode, and
- FOUR input ports named READ, READOFFSET, CONSUME, and CONSUMEOFFSET, all of integer mode.

**Output Ports:**
- an output port named OUTPUT whose mode is derived from family of INPUT'S mode as follows:
- the base type of OUTPUT is the same as the base type of INPUT, and the height of the OUTPUT mode is one more than the height of the INPUT mode.

READ, READOFFSET, CONSUME, and CONSUMEOFFSET may each be connected by a directed arc to the output port of a place with base type integer and height zero.

**Execution of pack may begin if:**
- Any place with an output data port that is connected to any of the ports READ, READOFFSET, CONSUME, or CONSUMEOFFSET is not empty.
- A directed arc connects INPUT to an output data port of a place **Q**, and **Q** satisfies content ≥ threshold, where threshold is determined as follows:
  - Define the five *Node Execution Parameters* (NEPs) to be *read*, *readoffset*, *consume*, *consumeoffset*, and *threshold*.
  - Let each of *read*, *readoffset*, *consume*, and *consumeoffset* be determined by the value read from the place with an output port connected to the respective input port READ, READOFFSET, CONSUME, and CONSUMEOFFSET.
  - If any of these ports is unconnected, then the following respective default values are assigned: *read* = 1, *readoffset* = 0, *consume = read*, *consumeoffset* = 0.
  - Put *threshold* = max (*read* + *readoffset*, *consume* + *consumeoffset*).
- If a directed arc connects OUTPUT to an input data port of a place **P**, then pack's execution will not result in **P's** content exceeding P's capacity

**Execution of pack is:**
- If OUTPUT is connected to an input port of a place **P**, a single token is produced to **P**.
- This token is a family of the values of the (*readoffset* + 1)th through (*readoffset* + *read*)th tokens in the token family of **Q**.
- The order of the elements in the family preserves the order of the corresponding tokens in **Q**.
- If OUTPUT is not connected to an *input port* of any *place*, then the *token* is discarded.
- The (*consumeoffset* + 1)th through (*consumeoffset* + *consume*)th tokens are consumed from **Q**.
- Let R be a place with an output data port that is connected to one of the ports READ, READOFFSET, CONSUME, or CONSUMEOFFSET. One token on R is consumed.

#### 5.3.3.2.2. Unpack

**Brief description:** Each *unpack transition* reads a token from its input place, the type of this token being a family of positive height and any base type. Each of the individual children in the family is then made into a token and produced to its output place. Each token produced is a family of height one less than the height of the token taken in.

**Detailed description:**

**Input Ports:**

- Each unpack transition has an input port called INPUT that takes families of any positive height and of any mode.

**Output Ports:**

- Two output ports called OUTPUT and PRODUCE. The mode of OUTPUT is the mode of a family member of INPUT. PRODUCE has integer mode.

Each of the three ports may be may be connected to an appropriate port of a place.

**Execution of an unpack transition may begin if:**

- A directed arc connects INPUT to an output data port of a place **P**, and **P** satisfies content ≥ 1.
- Let **T** be the token read from **P**. If a directed arc connects OUTPUT to an input data port of a place **Q**, then executing unpack will not result in the content of **Q** exceeding the capacity of **Q**.
- If a directed arc connects PRODUCE to an input data port of a place **R**, then executing unpack will not result in the content of **R** exceeding the capacity of **R**.

**Execution of an unpack transition is:**

- If OUTPUT is connected to an input port of a place **Q**, then each family member of **T** is produced to **Q**. The order of the tokens produced to **Q** preserves the order of the corresponding values in **T**.
- If OUTPUT is not connected to an input port of any place, then no token is written.
- If PRODUCE is connected to the input port of a place **R**, then a single integer token, whose value is content of **T**, is written to **R**.
- If PRODUCE is not connected to an input port of any place, then no token is written.
- One token is consumed from **P**.

## 5.4. Graph determinacy, a warning

This section gives a brief description of the concept of *determinacy* and how it can be managed in processing graphs. Consider a graph with a given sequence of tokens read at each of its graph input ports. There may exist different valid sequences of transition execution for the graph. Thus it is possible that the sequence of tokens produced at each output port of the graph may depend on the order in which the transitions execute. A graph is said to have determinacy if the sequence of tokens produced at each output port is determined solely by the sequence of tokens read at each input port. That means that the output sequence of tokens is independent of the order of transition execution and independent of the implementation of PGM that processes the graph.

Depending on the intended use, the user may wish to develop graphs that have determinacy. On the other hand, the user may wish to develop graphs that do not have determinacy while controlling that non-determinacy. Thus it is important to understand the factors that may lead to non-determinacy.

We call a place **P** determinate if **P** meets the following criteria:

- If **P** is a graph variable, then the input data port family of **P** is empty.
- **P** is a queue, then each of the associated input and output data port families has height zero or is empty.

We call a place **P** non-determinate if **P** is not determinate. A graph **G** will have determinacy if all its places are determinate.

If **P** is a non-determinate place, then there may be two transitions **T1** and **T2** connected to **P**, such that the order in which **T1** and **T2** execute determines the state of **P**. In other words, the values of tokens in **P** depend on the order of execution of **T1** and **T2**.

**Warning:** A particularly serious kind of non-determinacy arises when data obtained from a non-determinate place determines the value of a NEP in a pack transition. This may result in unrepeatable token values and unrepeatable numbers of tokens, resulting in a loss of synchronization between parallel pathways within the graph.

## 6 Graph construction

In a given system, there is at most one *main graph*. The main graph may have any number of included graphs. Each included graph may, in turn, have any number of included graphs. Henceforth in this section, unless otherwise stated, we use the word graph to mean either main graph or included graph.

Let **G** be a graph, let **N0** be a node in **G**, and let **P** be an input port or output port of **N0**. P may be connected to a port of another node **N1 ≠ N0** in **G**. Or **P** may be associated with a graph port **GP** of **G**. If **P** is associated with **GP**, then we say that **GP** is an alias of **P**. If **P** is associated with **GP**, then **P** may not be connected to any port of a node or included graph in **G**.

We say that **GP** has the same category, direction, and mode as its associated node port. This, if **N0** is a transition and **P** is an input port of **N0**, we say that **GP** is a graph input transition port of **G**. In a similar way, we may refer to a graph output transition port, graph input place port, or graph output place port of **G**.

Now let **H** be an included graph in **G**, and let **P** be a port of any node or included graph in **G**. It follows that **P** may be connected to a port **HP** of **H** if and only if the following conditions are met:

- **P** and **HP** have opposite category, i.e., one is a place port and the other is a transition port.
- **P** and **HP** have opposite direction, i.e., one is an input port and the other is an output port.
- **P** and **HP** have the same mode.

Let **P** be a port of a node or included graph **K** in **G**, and let **H≠K** be an included graph in **G**. In the construction of **G**, a directed arc may connect **P** with a graph port **HP** of **H**. However, connecting a directed arc between **P** and a node port of a node in **H** is not permitted.
All graph ports of the main graph are place ports.

Just as each port of a node is contained in a port family of the node, each port of a graph is contained in a port family of the graph. All ports in a port family must have the same mode.
In the construction of a graph, one specifies families of graph ports, families of nodes, and families of included graphs.

Each transition in a family of transitions must have been constructed from the same IP, which specifies the transition statement and the set of input and output port families of the transition. Each place in a family of places must have been constructed from the same IP, which specifies the kind of place (i.e., queue or graph variable) and its mode. Each included graph in a family of included graphs must have been constructed from the same IP, which specifies the node families and the set of input and output port families of the included graph.

Let **G** be a graph, let **N** be a family of nodes (or included graphs) $\mathbf{N}_i$ in **G** with height **H0**, and let $\mathbf{P}_i$ be a family of ports of $\mathbf{N}_i$. Then all port families $\mathbf{P}_i$ must have the same height, and all ports in every $\mathbf{P}_i$ must have the same mode. We define the port family **P** to be the family assembled from the family tree of **N** and the ports $\mathbf{P}_i$.

Association of node or included graph ports with graph ports must be by family. This means that a family of graph ports of a graph **G** is an alias of a family of ports of a node family or included graph family in **G**. More specifically, let **G** be a graph and let **GP** be a family of graph ports of **G**. Let **P** be a family of ports of a node family or included graph family in **G** that is assembled as described above. Then **P** may be associated with **GP** if and only if the following conditions are met:

- All the ports of **P** and of **GP** have the same category, direction, and mode, and
- **P** and **GP** have the same family tree.

When the main graph is constructed, all node families are instantiated as individual nodes. All included graphs are instantiated by execution of their IPs to create their respective nodes.

After graph construction, an implementation of PGM may construct a *flat graph* by resolving and eliminating all included graph port associations, together with family indices, into respective node ports and connecting node ports directly with node ports. In the flat graph, execution performance is enhanced by this elimination of intermediate included graph ports.

## 7 Command programs

The command program is an application specific program that serves a variety of different purposes.

If all of the transitions of the graph are blocked from starting an execution, the graph is said to be in a *suspended state*.

When a graph is created it is in a suspended state.

The command program may create one main graph and start it processing, or suspend execution. The command program may also write data into the graph's input place ports and read data from the graph's output place ports.

The command program can monitor and control the translation of data from external sources and introduce them as tokens into appropriate graph input place ports for graph processing. Examples of an external source are a sensor that measures observable phenomena, an operator or separate computer system, and a data storage medium like a disk file.

The command program can translate the tokens read at a graph output place port into appropriate form for delivery to external destinations. Examples of an external destination are a graphics display monitor, an operator or separate computer system via a message, and a data storage medium like a disk file.

The following sections describe the set of command program procedures to be used in building the graph and interacting with the graph. The command program procedures intended for building the graph are described first.

### 7.1. Instantiation procedures (IPs) and graph construction procedures (GCPs)

The command program constructs the main graph by calling its IP. The main graph's IP may call other IPs to construct the main graph's nodes and included graphs. Each included graph's IP may call still other IPs to construct its nodes and included graphs. Each graph IP may also call specific GCPs to perform other functions necessary for complete graph construction.

The GCPs are part of PGM. They have names that remain the same, regardless of what underlying high-level language is used. Moreover they may be called in any order, subject to the requirement that a variable must be defined before it can be used. A GCP must be called within the body of an IP, and it applies to one or more objects within the graph being constructed by that IP.

IPs are not canonical, their names are not part of PGM. There is no fixed “correct” way of writing them. There is no enforced calling sequence for them. We do provide a suggested sequence that is not enforced. The suggested calling sequence is the same as the sequence that our GUI editor will produce when it generates an IP.

Some of the GCPs have names of graphs, graph ports, or nodes as arguments. These names are character strings that are assigned as attributes of the respective graphs, graph ports, or nodes. These names do not refer to the variable names used in the source code to identify the respective entities.

#### 7.1.1. Ips

A formal IP call specifies the name of the IP and its formal input and output arguments.

GIPs are values used by the IP when constructing a node or graph to determine the topology and state of the node or graph. A GIP of a graph IP may be used to determine the number of times a loop is executed to construct a family of nodes or of included graphs. GIPs of a node or included graph IP may be used to specify parameters of a family tree for a family of ports, nodes, or included graphs. A GIP of a place IP may be used to determine the initial values of tokens in places of the graph. GIP may be passed as an argument of another IP.

A graph **G** consists of:

- All of its nodes, included graphs, directed arcs, and the initial tokens in the places of **G**.
- A possibly empty set of graph input ports and graph output ports of **G**. Each graph input port and graph output port of **G** identifies an unconnected port of a place in **G**. We refer to the graph input ports and graph output ports as graph ports. We refer to each graph port as a graph place port or graph transition port.

A program that can access **G** may use the graph place ports of **G** to introduce data into **G** for processing and for retrieving the resulting processed data from **G**.

- Our suggested formal inputs for an IP are:
  GIPs,
- Families of input ports,
- Families of output ports,
- Implementation specific parameters.

Each family of ports comprises both the family name and a specification for the family tree. This specification tells how to construct the family tree by using some or all of the assigned values of the GIPs.

The nodes and included graphs of a graph are internal to the graph. All other PGM entities (i.e., the command program and all nodes and included graphs not internal to the graph) are external to the graph. The interface between a given graph and its external entities make up the graph's exterior. This interface comprises the graph's input ports and output ports. We extend the notion of exterior to the graph's IP by defining the exterior of an IP to be the collection of formal inputs for that IP.

Our suggested output for an IP is a node or graph constructed by the IP.

### 7.1.2. GCPs

Each GCP call results in a specific action, which is a piece of the graph building process.

Each GCP causes an action that either aids in the construction of or modifies objects within a graph.

A GCP may be used to install a node **N** or an included graph **S** in a graph.

A GCP may be used to connect a directed arc between two ports of nodes or included graphs.

GCPs may be used to construct the family tree for one or more initial tokens of a place, for a family of nodes or included graphs, or for a family of ports of a node or graph.

GCPs may be used to construct tokens for initial values of places. The command program may call the same GCPs to construct a token for introduction to input place ports of the main graph.

#### 7.1.2.1. Connect two ports with a directed arc

**Function name:** connectPorts
**Input arguments:**

- family name of the node or included graph containing the output port,
- array of family indices to identify the node or included graph in the family,
- family name of the output port,
- array of family indices to identify the output port in the family,
- family name of the node or included graph containing the input port,
- array of family indices to identify the node or included graph in the family,
- family name of the input port,
- array of family indices to identify the input port in the family.

**Output arguments:**
None.

**Discussion:**
Let **NP** denote the output port and **N'P** denote the input port such that the modes of **NP** and **N'P** are the same, but of opposite category. Procedure connectPorts causes a directed arc to be formed in **G** from **NP** to **N'P**.

#### 7.1.2.2. Associate a family of node or included graph input ports with a family of graph

**Function name:** associateGraphInport
**Input arguments:**

- Input graph port family name,
- Family name of node or included graph containing the input port family to be associated,
- Family name of the input port family.

**Output arguments:**
None.

**Discussion:**

The input graph port family and the input port family of the node or included graph must have the same family tree. Each input graph port is associated with the input port of the respective node or included graph (i.e., with the same family indices).

#### 7.1.2.3. Associate a family of node or included graph output ports with a family of

**Function name:** associateGraphOutport
**Input arguments:**

- Family name of node or included graph containing the output port family to be associated,
- Family name of the output port family,
- Output graph port family name.

**Output arguments:**

- None.

### 7.1.3. Graph manipulation and interaction

The following sections describe additional procedures that may be called by a command program for purposes of manipulating graphs, and interacting with graphs.

The output graph port family and the output port family of the node or included graph must have the same family tree. Each output graph port is associated with the output port of the respective node or included graph (i.e., with the same family indices).

#### 7.1.3.1. Start execution of a main graph

**Function name:** startGraph
**Input arguments:**

- Newly created graph.

**Output arguments:**

- None.

**Discussion:**
Execution of this procedure enables all graph transitions.

#### 7.1.3.2. Suspend execution of a main graph

**Function name:** suspendGraph
**Input arguments:**

- The graph to be suspended.

**Output arguments:**

- None.

**Discussion:**
Execution of this procedure suspends all transitions in the graph, preventing every transition from starting or restarting an execution. No transition in the suspended graph is blocked from completing ongoing execution.

#### 7.1.3.3. Restart a suspended main graph

**Function name:** reviveGraph
**Input arguments:**

- The graph to be restarted.

**Output arguments:**
- None.

**Discussion:**
The graph must be in the suspended state. Execution of this procedure enables all graph transitions.

#### 7.1.3.4. Delete a main graph
**Function name:** Implementation specific - the high order language may have a specific function for deleting objects.
**Input arguments:**
- suspended graph to be deleted.

**Output arguments:**
- none

**Discussion:**
The graph must be in the suspended state. All transitions, places, and included graphs in the graph are deleted. .

#### 7.1.3.5. Get access to a main graph input place port
**Function name:** getInPort
**Input arguments:**
- The family name of the graph input place port,
- An array of family indices.

**Output arguments:**
- The desired graph input port.

**Discussion:**
The family name and the array of family indices identify the desired port
.

#### 7.1.3.6. Get access to a main graph output place port
**Function name:** getOutPort
**Input arguments:**
- The family name of the graph output place port,
- An array of family indices.

**Output arguments:**
- The desired graph output port.

**Discussion:**
The family name and the array of family indices identify the desired port.

#### 7.1.3.7. Write one token to the place associated with the main graph input place port
**Function name:** putToken
**Input arguments:**
- Graph input place port,
- A token.

**Output arguments:**
- None.

**Discussion:**
The result of successfully executing this function may not cause the place to exceed its capacity.

#### 7.1.3.8. Read one token from the place associated with the main graph output place port
**Function name:** getToken
**Input arguments:**
- Graph output port,

**Output arguments:**

- token.

The content of the place must be at least one for successful execution of this function.